Shared Memory Concurrent System Verification using Kronecker Algebra

Technical Report 183/1-155

Robert Mittermayr and Johann Blieberger

Institute of Computer-Aided Automation, TU Vienna, Austria

Abstract. The verification of multithreaded software is still a challenge. This comes mainly from the fact that the number of thread interleavings grows exponentially in the number of threads. The idea that thread interleavings can be studied with a matrix calculus is a novel approach in this research area. Our sparse matrix representations of the program are manipulated using a lazy implementation of Kronecker algebra. One goal is the generation of a data structure called *Concurrent Program Graph* (CPG) which describes all possible interleavings and incorporates synchronization while preserving completeness. We prove that CPGs in general can be represented by sparse adjacency matrices. Thus the number of entries in the matrices is linear in their number of lines. Hence efficient algorithms can be applied to CPGs. In addition, due to synchronization only very small parts of the resulting matrix are actually needed, whereas the rest is unreachable in terms of automata. Thanks to the lazy implementation of the matrix operations the unreachable parts are never calculated. This speeds up processing significantly and shows that this approach is very promising.
Various applications including data flow analysis can be performed on CPGs. Furthermore, the structure of the matrices can be used to prove properties of the underlying program for an arbitrary number of threads. For example, deadlock freedom is proved for a large class of programs.

1 Introduction

With the advent of multi-core processors scientific and industrial interest focuses on the verification of multithreaded applications. The scientific challenge comes from the fact that the number of thread interleavings grows exponentially in a program's number of threads. All state-of-the-art methods, such as model checking, suffer from this so-called *state explosion problem*. The idea that thread interleavings can be studied with a matrix calculus is new in this research area. We are immediately able to support conditionals, loops, and synchronization. Our sparse matrix representations of the program are manipulated using a lazy implementation of Kronecker algebra. Similar to [3] we describe synchronization by Kronecker products and thread interleavings by Kronecker sums. One goal is the generation of a data structure called *Concurrent Program Graph* (CPG) which describes all possible interleavings and incorporates synchronization while

preserving completeness. Similar to CFGs for sequential programs, CPGs may serve as an analogous graph for concurrent systems. We prove that CPGs in general can be represented by sparse adjacency matrices. Thus the number of entries in the matrices is linear in their number of lines.

In the worst-case the number of lines increases exponentially in the number of threads. Especially for concurrent programs containing synchronization this is very pessimistic. For this case we show that the matrix contains nodes and edges unreachable from the entry node.

We propose two major optimizations. First, if the program contains a lot of synchronization, only a very small part of the CPG is reachable. Our lazy implementation of the matrix operations computes only this part (cf. Subsect. 3.6). Second, if the program has only little synchronization, many edges not accessing shared variables will be present, which are reduced during the output process of the CPG (cf. Subsect. 3.7). Both optimizations speed up processing significantly and show that this approach is very promising.

We establish a framework for analyses of multithreaded shared memory concurrent systems which forms a basis for analyses of various properties. Different techniques including dataflow analysis (e.g. [23–25, 14]) and model checking (e.g. [6, 9] to name only a few) can be applied to the generated *Concurrent Program Graphs* (CPGs) defined in Section 3. Furthermore, the structure of the matrices can be used to prove properties of the underlying program for an arbitrary number of threads. For example in this paper, deadlock freedom is proved for p-v-symmetric programs.

Theoretical results such as [21] state that synchronization-sensitive and context-sensitive analysis is impossible even for the simplest analysis problems. Our system model differs in that it supports subprograms only via inlining and recursions are impossible.

The outline of our paper is as follows. In Section 2 control flow graphs, edge splitting, and Kronecker algebra are introduced. Our model of concurrency, its properties, and important optimizations like our lazy approach are presented in Section 3. In Section 4 we give a client-server example with 32 clients showing the efficiency of our approach. For a matrix with a potential order of 10^{15} our lazy approach delivers the result in 0.43s. Section 5 demonstrates how deadlock freedom is proved for p-v-symmetric programs with an arbitrary number of threads. An example for detecting a data race is given in Section 6. Section 7 is devoted to an empirical analysis. In Section 8 we survey related work. Finally, we draw our conclusion in Section 9.

2 Preliminaries

2.1 Overview

We model shared memory concurrent systems by threads which use semaphores for synchronization. Threads and semaphores are represented by control flow graphs (CFGs). Edge Splitting has to be applied to the edges of thread CFGs

that access more than one shared variable. Edge splitting is straight forward and is described in Subsect. 2.3. The resulting Refined CFGs (RCFGs) are represented by adjacency matrices. These matrices are then manipulated by Kronecker algebra. We assume that the edges of CFGs are labeled by elements of a semiring. Details follow in this subsection. Similar definitions and further properties can be found in [16].

Semiring $\langle \mathcal{L}, +, \cdot, 0, 1 \rangle$ consists of a set of labels $\mathcal{L}$, two binary operations $+$ and $\cdot$, and two constants 0 and 1 such that

1. $\langle \mathcal{L}, +, 0 \rangle$ is a commutative monoid,
2. $\langle \mathcal{L}, \cdot, 1 \rangle$ is a monoid,
3. $\forall l_1, l_2, l_3 \in \mathcal{L} : l_1 \cdot (l_2 + l_3) = l_1 \cdot l_2 + l_1 \cdot l_3$ and $(l_1 + l_2) \cdot l_3 = l_1 \cdot l_3 + l_2 \cdot l_3$ hold and
4. $\forall l \in \mathcal{L} : 0 \cdot l = l \cdot 0 = 0$.

Intuitively, our semiring is a unital ring without subtraction. For each $l \in \mathcal{L}$ the usual rules are valid, e.g., $l + 0 = 0 + l = l$ and $1 \cdot l = l \cdot 1 = l$. In addition we equip our semiring with the unary operation $*$. For each $l \in \mathcal{L}$, l^* is defined by $l^* = \sum_{j \geq 0} l^j$, where $l^0 = 1$ and $l^{j+1} = l^j \cdot l = l \cdot l^j$ for $j \geq 0$. Our set of labels $\mathcal{L}$ is defined by $\mathcal{L} = \mathcal{L}_{\mathrm{V}} \cup \mathcal{L}_{\mathrm{S}}$, where $\mathcal{L}_{\mathrm{V}}$ is the set of non-synchronization labels and $\mathcal{L}_{\mathrm{S}}$ is the set of labels representing semaphore calls. The sets $\mathcal{L}_{\mathrm{V}}$ and $\mathcal{L}_{\mathrm{S}}$ are disjoint. The set $\mathcal{L}_{\mathrm{S}}$ itself consists of two disjoint sets $\mathcal{L}_{\mathrm{S}_p}$ and $\mathcal{L}_{\mathrm{S}_v}$. The first denotes the set of labels referring to P-calls, whereas the latter refers to V-calls of semaphores.

Examples for semirings include regular expressions (cf. [26]) which can be used for performing dataflow analysis.

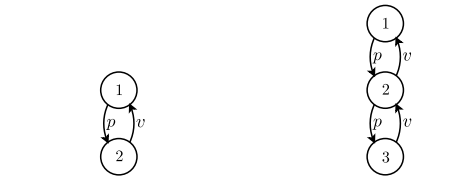
2.2 Control Flow Graphs

A *Control Flow Graph* (CFG) is a directed labeled graph defined by $G = \langle V, E, n_e \rangle$ with a set of nodes V, a set of directed edges $E \subseteq V \times V$, and a so-called *entry* node $n_e \in V$. We require that each $n \in V$ is reachable through a sequence of edges from n_e. Nodes can have at most two outgoing edges. Thus the maximum number of edges in CFGs is $2\,|V|$. We will use this property later.

Usually CFG nodes represent basic blocks (cf. [1]). Because our matrix calculus manipulates the edges we need to have basic blocks on the edges.[1] Each edge $e \in E$ is assigned a basic block b. In this paper we refer to them as edge labels as defined in the previous subsection. To keep things simple we use edges, their labels and the corresponding entries of the adjacency matrices synonymously.

In order to model synchronization we use semaphores. The corresponding edges typically have labels like p_1 and v_1, where p_x and $v_x \in \mathcal{L}_{\mathrm{S}}$. Usually two or more distinct thread CFGs refer to the same semaphore to perform synchronization. The other labels are elements from $\mathcal{L}_{\mathrm{V}}$. The operations on the basic blocks are $\cdot$, $+$, and $*$ from the semiring defined above (cf. [26]). Intuitively, $\cdot$, $+$, and $*$ model consecutive basic blocks, conditionals, and loops, respectively.

[1] We chose the incoming edges.



(a) Binary Semaphore (b) Counting Semaphore

Fig. 1. Semaphores

In Fig. 1(a) and 1(b) a binary and a counting semaphore are depicted. The latter allows two threads to enter at the same time. In a similar way it is possible to construct semaphores allowing n non-blocking P-calls.

2.3 Edge Splitting

A basic block consists of multiple consecutive statements without jumps. For our purpose we need a finer granularity as we would have with basic blocks alone. To achieve the required granularity we need to split edges. Shared variable accesses and semaphore calls may occur in basic blocks. For both it is necessary to split edges. This ensures a representation of possible context switches in a manner exact enough for our purposes. We say "exact enough" because by using basic blocks together with the above refinement, we already have coarsened the analysis compared to the possibilities on statement-level. Furthermore we do not lose any information required for the completeness of our approach. Anyway, applying this procedure to a CFG, i.e. splitting edges in a CFG, results in a *Refined Control Flow Graph* (RCFG).

Let $\mathcal{V}$ be the set of shared variables. In addition, let a shared variable $v \in \mathcal{V}$ be a volatile variable located in the shared memory which is accessed by two or more threads. Splitting an edge depends on the number of shared variables accessed in the corresponding basic block. For edge e this number is being referred to as $\mathrm{NSV}(e)$. In the same way we refer to $\mathrm{NSV}(b)$ as the number of shared variables accessed in basic block b. If $\mathrm{NSV}(e) > 1$, edge splitting has to be applied to edge e; the edge is used unchanged otherwise.

If edge splitting has to be applied to edge e which has basic block b assigned and $\mathrm{NSV}(b) = k$ then the basic blocks $b_1, \ldots, b_k$ represent the subsequent parts of b in such a way that $\forall b_i : \mathrm{NSV}(b_i) = 1$, where $1 \leq i \leq k$. Edges e_j get assigned basic block b_j, where $1 \leq j \leq k$. In Fig. 2 the splitting of an edge with basic block b and $\mathrm{NSV}(b) = k$ is depicted.

For semaphore calls (e.g. p_1 and v_1) edge splitting is required in a similar fashion. In contrast to shared variable accesses we require that semaphore calls have to be the only statement on the corresponding edge. The remaining consec-

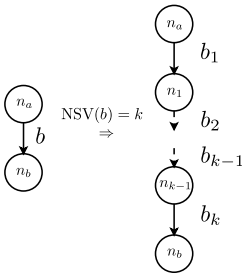


Fig. 2. Edge Splitting for Shared Variable Accesses

utive parts of the basic block are situated on the previous and succeeding edges, respectively.[2]

The effects of edge splitting for shared variables and semaphore calls can be seen in the data race example given in Section 6. Each RCFG depicted in Fig. 12 is constructed out of one basic block (cf. Fig. 11).

Note that edge splitting ensures that we can model the minimal required context switches. The semantics of a concurrent programming language allows usually more. For example consider an edge in a RCFG containing two consecutive statements, where both do not access shared variables. A context switch may happen in between. However, this additional interleaving does not provide new information. Hence our approach provides the minimal number of context switches.

Without loss of generality we assume that the statements in each basic block are atomic. Thus, we assume while executing a statement, context switching is impossible. In RCFGs the finest possible granularity is at statement-level. If, according to the program's semantic, atomic statements may access two or more shared variables, then we make an exception to the above rule and allow two or more shared variable accesses on a single edge. Such edges have at most one atomic statement in their basic block. The Kronecker sum (which is introduced in the next subsection) ensures that all interleavings are generated correctly.

2.4 Synchronization and Generating Interleavings with Kronecker Algebra

Kronecker product and Kronecker sum form Kronecker algebra. In the following we define both operations, state properties, and give examples. In addition, for the Kronecker sum we prove a property which we call *Mixed Sum Rule*.

We define the set of matrices $\mathcal{M} = \{M = (m_{i,j}) \mid m_{i,j} \in \mathcal{L}\}$. In the remaining parts of this paper only matrices $M \in \mathcal{M}$ will be used, except where stated explicitly. Let $o(M)$ refer to the order[3] of matrix $M \in \mathcal{M}$. In addition we will use n-by-n zero matrices $Z_n = (z_{i,j})$, where $\forall i, j : z_{i,j} = 0$.

[2] Note that edges representing a call to a semaphore are not considered to access shared variables.

[3] A k-by-k matrix is known as square matrix of order k.

Definition 1 (Kronecker product). *Given a m-by-n matrix A and a p-by-q matrix B, their* Kronecker product *denoted by $A \otimes B$ is a mp-by-nq block matrix defined by*

$$A \otimes B = \begin{pmatrix} a_{1,1}B & \cdots & a_{1,n}B \\ \vdots & \ddots & \vdots \\ a_{m,1}B & \cdots & a_{m,n}B \end{pmatrix}.$$

Example 1.

Let $A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$ and $B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$. The Kronecker product $C = A \otimes B$ is given by

$$\begin{pmatrix} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & a_{1,1}b_{1,3} & a_{1,2}b_{1,1} & a_{1,2}b_{1,2} & a_{1,2}b_{1,3} \\ a_{1,1}b_{2,1} & a_{1,1}b_{2,2} & a_{1,1}b_{2,3} & a_{1,2}b_{2,1} & a_{1,2}b_{2,2} & a_{1,2}b_{2,3} \\ a_{1,1}b_{3,1} & a_{1,1}b_{3,2} & a_{1,1}b_{3,3} & a_{1,2}b_{3,1} & a_{1,2}b_{3,2} & a_{1,2}b_{3,3} \\ a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & a_{2,1}b_{1,3} & a_{2,2}b_{1,1} & a_{2,2}b_{1,2} & a_{2,2}b_{1,3} \\ a_{2,1}b_{2,1} & a_{2,1}b_{2,2} & a_{2,1}b_{2,3} & a_{2,2}b_{2,1} & a_{2,2}b_{2,2} & a_{2,2}b_{2,3} \\ a_{2,1}b_{3,1} & a_{2,1}b_{3,2} & a_{2,1}b_{3,3} & a_{2,2}b_{3,1} & a_{2,2}b_{3,2} & a_{2,2}b_{3,3} \end{pmatrix}.$$

As stated in [18] the Kronecker product is also being referred to as *Zehfuss product* or *direct product of matrices* or *matrix direct product.* [4]

In the following we list some basic properties of the Kronecker product. Proofs and additional properties can be found in [2, 10, 7, 11]. Let A, B, C, and D be matrices. The Kronecker product is noncommutative because in general $A \otimes B \neq B \otimes A$. It is permutation equivalent because there exist permutation matrices P and Q such that $A \otimes B = P(B \otimes A)Q$. If A and B are square matrices, then $A \otimes B$ and $B \otimes A$ are even permutation similar, i.e., $P = Q^T$. The product is associative as

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C. \tag{1}$$

In addition, the Kronecker product distributes over $+$, i.e.,

$$A \otimes (B + C) = A \otimes B + A \otimes C, \tag{2}$$
$$(A + B) \otimes C = A \otimes C + B \otimes C. \tag{3}$$

Hence for example $(A + B) \otimes (C + D) = A \otimes C + B \otimes C + A \otimes D + B \otimes D$.

The Kronecker product allows to model synchronization (cf. Subsect. 3.2).

Definition 2 (Kronecker sum). *Given a matrix A of order m and matrix B of order n, their* Kronecker sum *denoted by $A \oplus B$ is a matrix of order mn defined by*

$$A \oplus B = A \otimes I_n + I_m \otimes B,$$

[4] Knuth notes in [15] that Kronecker never published anything about it. Zehfuss was actually the first publishing it in the 19th century [27]. He proved that $\det(A \otimes B) = \det^n(A) \cdot \det^m(B)$, if A and B are matrices of order m and n and entries from the domain of real numbers, respectively.

where I_m and I_n denote identity matrices[5] *of order m and n, respectively.*

This operation must not be confused with the direct sum of matrices, group direct product or direct product of modules for which the symbol $\oplus$ is used too. By calculating the Kronecker sum of the adjacency matrices of two graphs the adjacency matrix of the Cartesian product graph [12] is computed (cf. [15]).

Example 2. We use matrices A and B from Ex. 1. The Kronecker sum $A \oplus B$ is given by

$$
\begin{aligned}
&A \otimes I_3 + I_2 \otimes B = \\
&\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} = \\
&\begin{pmatrix} a_{1,1} & 0 & 0 & a_{1,2} & 0 & 0 \\ 0 & a_{1,1} & 0 & 0 & a_{1,2} & 0 \\ 0 & 0 & a_{1,1} & 0 & 0 & a_{1,2} \\ a_{2,1} & 0 & 0 & a_{2,2} & 0 & 0 \\ 0 & a_{2,1} & 0 & 0 & a_{2,2} & 0 \\ 0 & 0 & a_{2,1} & 0 & 0 & a_{2,2} \end{pmatrix} + \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & 0 & 0 & 0 \\ b_{2,1} & b_{2,2} & b_{2,3} & 0 & 0 & 0 \\ b_{3,1} & b_{3,2} & b_{3,3} & 0 & 0 & 0 \\ 0 & 0 & 0 & b_{1,1} & b_{1,2} & b_{1,3} \\ 0 & 0 & 0 & b_{2,1} & b_{2,2} & b_{2,3} \\ 0 & 0 & 0 & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} = \\
&\begin{pmatrix} a_{1,1}+b_{1,1} & b_{1,2} & b_{1,3} & a_{1,2} & 0 & 0 \\ b_{2,1} & a_{1,1}+b_{2,2} & b_{2,3} & 0 & a_{1,2} & 0 \\ b_{3,1} & b_{3,2} & a_{1,1}+b_{3,3} & 0 & 0 & a_{1,2} \\ a_{2,1} & 0 & 0 & a_{2,2}+b_{1,1} & b_{1,2} & b_{1,3} \\ 0 & a_{2,1} & 0 & b_{2,1} & a_{2,2}+b_{2,2} & b_{2,3} \\ 0 & 0 & a_{2,1} & b_{3,1} & b_{3,2} & a_{2,2}+b_{3,3} \end{pmatrix}.
\end{aligned}
$$

In the following we list basic properties of the Kronecker sum of matrices A, B, and C. Additional properties can be found in [20] or are proved in this paper. The Kronecker sum is noncommutative because for element-wise comparison in general $A \oplus B \neq B \oplus A$. Anyway it essentially commutes because from a graph point of view, the graphs represented by matrices $A \oplus B$ and $B \oplus A$ are structurally isomorphic.

Now we state a property of the Kronecker sum which we call *Mixed Sum Rule*.

Lemma 1. *Let the matrices A and C have order m and B and D have order n. Then we call*

$$(A \oplus B) + (C \oplus D) = (A + C) \oplus (B + D)$$

the Mixed Sum Rule.

Proof. By using Eqs. (2) and (3) and Def. 2 we get $(A \oplus B) + (C \oplus D) = A \otimes I_n + I_m \otimes B + C \otimes I_n + I_m \otimes D = (A+C) \otimes I_n + I_m \otimes (B+D) = (A+C) \oplus (B+D)$. □

[5] The identity matrix I_n is a n-by-n matrix with ones on the main diagonal and zeros elsewhere.

For example let the matrices A and B be written as $A = \sum_{i \in I} A_i$ and $B = \sum_{j \in J} B_j$, respectively. In addition, let the sets I and J have the same number of elements, i.e., $|I| = |J|$. By using the mixed sum rule we can write $A \oplus B = \sum_{i \in I, j \in J} A_i \oplus B_j$.

We will frequently use the Mixed Sum Rule from now on without further notice.

The Kronecker sum is also associative, as $(A \oplus B) \oplus C$ and $A \oplus (B \oplus C)$ are equal.

Lemma 2. *Kronecker sum is associative.*

Proof. In the following we will use $I_m \otimes I_n = I_{m.n}$. Note that Z denotes zero matrices. We have

$$\begin{aligned}
A \oplus (B \oplus C) &= A \oplus (B \otimes I_{o(C)} + I_{o(B)} \otimes C) \\
\{\text{adding } Z_{o(A)}\} &= (A + Z_{o(A)}) \oplus (B \otimes I_{o(C)} + I_{o(B)} \otimes C) \\
\{\text{Lemma 1}\} &= (A \oplus (B \otimes I_{o(C)})) + (Z_{o(A)} \oplus (I_{o(B)} \otimes C)) \\
\{\text{Eq.(1), Def.2}\} &= (A \oplus (B \otimes I_{o(C)})) + I_{o(A)} \otimes I_{o(B)} \otimes C \\
\{\text{ass.+, Def.2}\} &= A \otimes I_{o(B).o(C)} + I_{o(A)} \otimes B \otimes I_{o(C)} + \\
&\quad I_{o(A).o(B)} \otimes C \\
\{\text{comm. of } +\} &= A \otimes I_{o(B)} \otimes I_{o(C)} + I_{o(A).o(B)} \otimes C + \\
&\quad I_{o(A)} \otimes B \otimes I_{o(C)} \\
\{\text{Def. 2}\} &= ((A \otimes I_{o(B)}) \oplus C) + I_{o(A)} \otimes B \otimes I_{o(C)} \\
\{\text{Def. 2}\} &= ((A \otimes I_{o(B)}) \oplus C) + ((I_{o(A)} \otimes B) \oplus Z_{o(C)}) \\
\{\text{Lemma 1}\} &= (A \otimes I_{o(B)} + I_{o(A)} \otimes B) \oplus (C + Z_{o(C)}) \\
\{\text{rm. } Z_{o(C)}\} &= (A \otimes I_{o(B)} + I_{o(A)} \otimes B) \oplus C \\
\{\text{Def. 2}\} &= (A \oplus B) \oplus C.
\end{aligned}$$

□

The associativity properties of the operations $\otimes$ and $\oplus$ imply that the k-fold operations

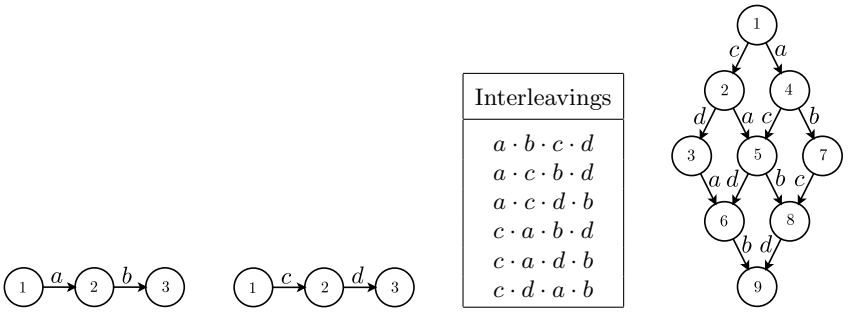
$$\bigotimes_{i=1}^{k} A_i \quad \text{and} \quad \bigoplus_{i=1}^{k} A_i$$

are well defined.

Note that Kronecker sum calculates all possible interleavings (see e.g. [17] for a proof). Note that this is true even for general CFGs including conditionals and loops. The following example illustrates interleaving of threads and how Kronecker sum handles it.

Example 3. Let the matrices C and D be defined as follows:

$$C = \begin{pmatrix} 0 & a & 0 \\ 0 & 0 & b \\ 0 & 0 & 0 \end{pmatrix} \qquad D = \begin{pmatrix} 0 & c & 0 \\ 0 & 0 & d \\ 0 & 0 & 0 \end{pmatrix}.$$



(a) C (b) D (c) Interleavings (d) $C \oplus D$

Fig. 3. A Simple Example

The graph corresponding to matrix C is depicted in Fig. 3(a), whereas the graph of matrix D is shown in Fig. 3(b). The regular expressions associated to the CFGs are $a \cdot b$ and $c \cdot d$, respectively. All possible interleavings by executing C and D in an interleavings semantics are shown in Fig. 3(c). In Fig. 3(d) the graph represented by the adjacency matrix $C \oplus D$ is depicted. It is easy to see that all possible interleavings are generated correctly.

3 Concurrent Program Graphs

Our system model consists of a finite number of threads and a finite number of semaphores. Both, threads and semaphores, are represented by CFGs. The CFGs are stored in form of adjacency matrices. The matrices have entries which are referred to as labels $l \in \mathcal{L}$ as defined in Subsect. 2.1. Let $\mathcal{S}$ and $\mathcal{T}$ be the sets of adjacency matrices representing semaphores and threads, respectively. The matrices are manipulated by using Kronecker algebra. Similar to [3] we describe synchronization by Kronecker products and thread interleavings by Kronecker sums. Note that higher synchronization features of programming languages such as Ada's rendezvous can be simulated by our system model as the runtime system uses semaphores provided by the operating systems to implement them.

Formally, the system model consists of the tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where

- $\mathcal{T}$ is the set of RCFG adjacency matrices describing threads,
- $\mathcal{S}$ is the set of CFG adjacency matrices describing semaphores, and
- $\mathcal{L}$ is the set of labels out of the semiring defined in Subsect. 2.1. The labels in $T \in \mathcal{T}$ are elements of $\mathcal{L}$, whereas the labels in $S \in \mathcal{S}$ are elements of $\mathcal{L}_{\mathrm{S}}$.

A *C*oncurrent Program Graph (CPG) is a graph $C = \langle V, E, n_e \rangle$ with a set of nodes V, a set of directed edges $E \subseteq V \times V$, and a so-called *entry* node $n_e \in V$. The sets V and E are constructed out of the elements of $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$. Details on how we generate the sets V and E follow in the next subsections. Similar to RCFGs the edges of CPGs are labeled by $l \in \mathcal{L}$. Assuming without loss of generality that each thread has an entry node with index 1 in its adjacency matrix $t \in \mathcal{T}$, then the entry node of the generated CPG has index 1, too.

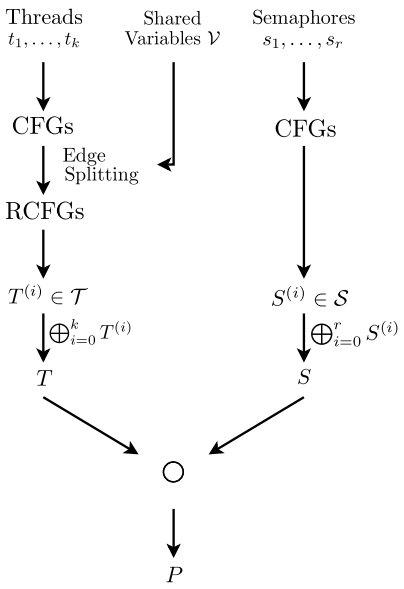


Fig. 4. Overview

In Fig. 4 an overview of our approach is given. As described in Subsect. 2.3 the set of shared variables $\mathcal{V}$ is used to generate $\mathcal{T}$.

3.1 Generating a Concurrent Program's Matrix

Let $T^{(i)} \in \mathcal{T}$ and $S^{(i)} \in \mathcal{S}$ refer to the matrices representing thread i and semaphore i, respectively. Let $M = (m_{i,j}) \in \mathcal{M}$. In addition, we define the matrix M_l as the matrix with entries of M equal to l and zeros elsewhere:

$$M_l = (m_{l;i,j}), \text{ where } m_{l;i,j} = \begin{cases} l \text{ if } m_{i,j} = l, \\ 0 \text{ otherwise.} \end{cases}$$

We obtain the matrix representing the k interleaved threads as

$$T = \bigoplus_{i=1}^{k} T^{(i)}, \text{ where } T^{(i)} \in \mathcal{T}.$$

According to Fig. 1 we have for the binary and the counting semaphore an adjacency matrix of order two and three, respectively. If we assume that the ith and the jth semaphore, where $1 \leq i, j \leq r$, are a binary and a counting semaphore, respectively, then we get the following adjacency matrices.

$$S^{(i)} = \begin{pmatrix} 0 & p_i \\ v_i & 0 \end{pmatrix} \text{ and } S^{(j)} = \begin{pmatrix} 0 & p_j & 0 \\ v_j & 0 & p_j \\ 0 & v_j & 0 \end{pmatrix}$$

In a similar fashion we can model counting semaphores of higher order.

The matrix representing the r interleaved semaphores is given by

$$S = \bigoplus_{i=1}^{r} S^{(i)}, \text{ where } S^{(i)} \in \mathcal{S}.$$

The adjacency matrix representing program $\mathcal{P}$ referred to as P is defined as

$$P = T \circ S = \sum_{l \in \mathcal{L}_{\mathrm{S}}} (T_l \otimes S_l) + \sum_{l \in \mathcal{L}_{\mathrm{V}}} (T_l \oplus S_l)\,. \tag{4}$$

When applying the Kronecker product to semaphore calls we follow the rules $v_x \cdot v_x = v_x$ and $p_x \cdot p_x = p_x$.

In Subsect. 3.5 we describe how the $\circ$-operation can be implemented efficiently.

3.2 $\circ$-Operation and Synchronization

Lemma 3. *Let $T = \bigoplus_{i=1}^{k} T^{(i)}$ be the matrix representing k interleaved threads and let S be a binary semaphore. Then $T \circ S$ correctly models synchronization of T with semaphore S.*[6]

Proof. First we observe that

1. the first term in the definition of Eq. (4) replaces
   - each p in matrix T with $\begin{pmatrix} 0 & p \\ 0 & 0 \end{pmatrix}$ and
   - each v in matrix T with $\begin{pmatrix} 0 & 0 \\ v & 0 \end{pmatrix}$,
2. the second term replaces each $m \in \mathcal{L}_{\mathcal{V}}$ with $\begin{pmatrix} m & 0 \\ 0 & m \end{pmatrix}$, and
3. both terms replace each 0 by $\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$.

According to the replacements above the order of matrix $T \circ S$ has doubled compared to T.

Now, consider the paths in the automaton underlying T described by the regular expression

$$\pi = \left(\sum\nolimits_{m \in \mathcal{L}_{\mathcal{V}}} m\right)^{*} \left(p \left(\sum\nolimits_{m \in \mathcal{L}_{\mathcal{V}}} m\right)^{*} v \left(\sum\nolimits_{m \in \mathcal{L}_{\mathcal{V}}} m\right)^{*}\right)^{*}.$$

By the observations above it is easy to see that paths containing π are present in $T \circ S$. On the other hand, paths not containing π are no more present in $T \circ S$. Thus the semaphore operations always occur in (p, v) pairs in all paths in $T \circ S$. This, however, exactly mirrors the semantics of synchronization via a semaphore. □

[6] Note that we do not make assumptions concerning the structure of T.

Generalizing Lemma 3, it is easy to see that the synchronization property is also correctly modeled if we replace the binary semaphore by one which allows more than one thread to enter it. In addition, the synchronization property is correctly modeled even if more than one semaphore is present on the right-hand side of $T \circ S$.

As a byproduct the proof of Lemma 3 shows the following corollary.

Corollary 1. *If the program modeled by $T \circ S$ contains a deadlock, then the matrix $T \circ S$ will contain a zero line ℓ. Node ℓ in the corresponding automaton is no final node and does not have successors.*

Thus deadlocks show up in CPGs as a pure structural property of the underlying graphs. Nevertheless, false positives may occur. From a static point of view, a deadlock is possible while conditions exclude this case at runtime. Our approach delivers a path to a deadlock in any case. Nevertheless, our approach of finding deadlocks is complete. If it states deadlock freedom, then the program under test is certainly deadlock free.

A further consequence of Lemma 3 is that after applying the $\circ$-operation only a small part of the underlying automata can be reached from its entry node. This allows for optimizations discussed later.

3.3 Unreachable Parts Caused by Synchronization

In this subsection we show that synchronization causes unreachable parts. As an example consider Fig. 5. The program consists of two threads, namely T_1 and T_2. The RCFGs of the threads are shown in Fig. 5(a) and Fig. 5(b). The used semaphore is a binary semaphore similar to Fig. 1(a). Its operations are referred to as p_1 and v_1. We denote a P and V-call to semaphore x of thread t as $t.p_x$ and $t.v_x$, respectively. T_1 and T_2 access the same shared variable in a and b, respectively. The semaphore is used to ensure that a and b are accessed mutually exclusively. Note that a and b may actually be subgraphs consisting of multiple nodes and edges.

For the example we have the matrices

$$T_1 = \begin{pmatrix} 0 & p_1 & 0 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & v_1 \\ 0 & 0 & 0 & 0 \end{pmatrix}, T_2 = \begin{pmatrix} 0 & p_1 & 0 & 0 \\ 0 & 0 & b & 0 \\ 0 & 0 & 0 & v_1 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and } S = \begin{pmatrix} 0 & p_1 \\ v_1 & 0 \end{pmatrix}.$$

Then we obtain the matrix $T = T_1 \oplus T_2$, a matrix of order 16, consisting of the submatrices defined above and zero matrices of order four (instead of Z_4 simply denoted by 0) as follows.

$$T = \begin{pmatrix} T_2 & p_1 \cdot I_4 & 0 & 0 \\ 0 & T_2 & a \cdot I_4 & 0 \\ 0 & 0 & T_2 & v_1 \cdot I_4 \\ 0 & 0 & 0 & T_2 \end{pmatrix}$$

In order to enable a concise presentation of $T \circ S$ we define the matrices

$$U = \begin{pmatrix} 0&0&0&p_1&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&b&0&0&0 \\ 0&0&0&0&0&b&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&v_1&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}, V = \begin{pmatrix} 0&p_1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&p_1&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&p_1&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&p_1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix},$$

$$W = a \cdot I_8, \text{ and } X = \begin{pmatrix} 0&0&0&0&0&0&0&0 \\ v_1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&v_1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&v_1&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&v_1&0 \end{pmatrix} \text{ of order 8.}$$

Then we obtain the matrix $T \circ S$, a matrix of order 32, consisting of the submatrices defined above and zero matrices of order eight (instead of Z_8 simply denoted by 0) as follows.

$$T \circ S = \begin{pmatrix} U & V & 0 & 0 \\ 0 & U & W & 0 \\ 0 & 0 & U & X \\ 0 & 0 & 0 & U \end{pmatrix}.$$

The generated CPG is depicted in Fig. 5(c). The resulting adjacency matrix has order 32, whereas the resulting CPG consists only of 12 nodes and 12 edges. Large parts (20 nodes and 20 edges) are unreachable from the entry node. In Fig. 6 these unreachable parts are depicted.

In general, unreachable parts exist if a concurrent program contains synchronization. If a program contains a lot of synchronization the reachable parts may be very small. This observation motivates the lazy implementation described in Subsect. 3.6.

3.4 Properties of the Resulting Adjacency Matrix

In this subsection we prove interesting properties of the resulting matrices.

A short calculation shows that the Kronecker sum in general generates at most $mn^2 + nm^2 - nm$ non-zero entries.[7] Stated the other way, at least $(mn)^2 - mn^2 - nm^2 + mn$ entries are zero. We will see that CFGs and RCFGs contain even more zero entries. We will prove that for this case the number of edges is in $O(mn)$. Thus, the number of edges is linear in the order of the resulting adjacency matrix.

[7] Assuming the corresponding matrices have an order of m and n, respectively.

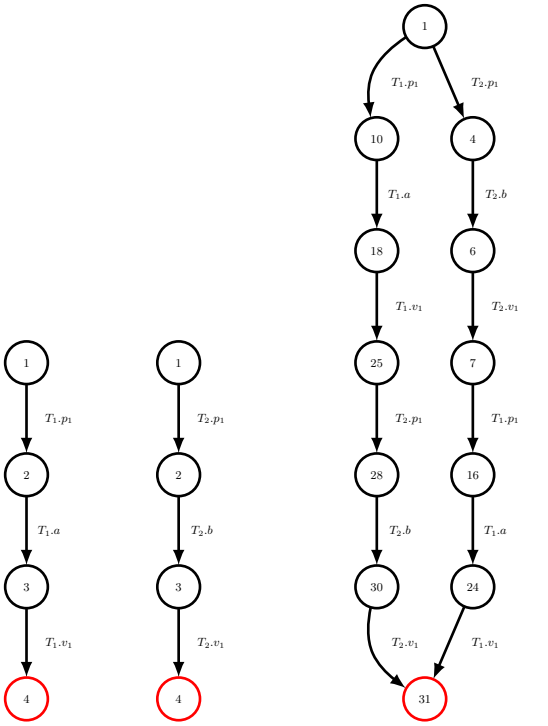


Fig. 5. Mutual Exclusion Example

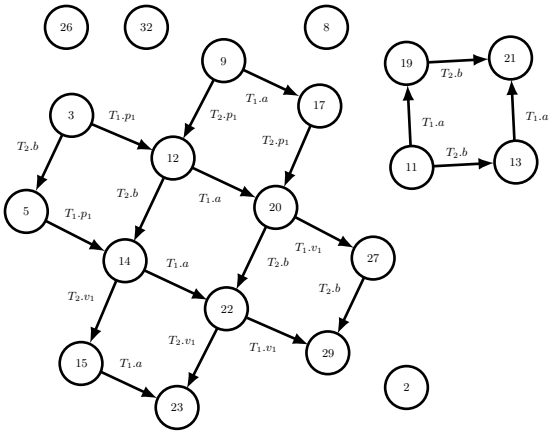


Fig. 6. Unreachable Parts of the Mutual Exclusion Example

Lemma 4 (Maximum Number of Nodes). *Given a program $\mathcal{P}$ consisting of $k > 0$ threads $(t_1, t_2, \ldots, t_k)$, where each t_i has n nodes in its RCFG, the number of nodes in $\mathcal{P}$'s adjacency matrix P is bounded from above by n^k.*

Proof. This follows immediately from the definitions of $\otimes$ and $\oplus$. For both the order of the resulting matrix is given by the multiplication of the orders of the input matrices. □

Definition 3. *Let $M = (m_{i,j}) \in \mathcal{M}$. We denote the number of non-zero entries by $||M|| = |\{m_{i,j} \mid m_{i,j} \neq 0\}|$.*

For a RCFG with n nodes it is easy to see that it contains at most $2n$ edges.

Lemma 5 (Maximum Number of Entries). *Let a program represented by $M_k \in \mathcal{M}$ consisting of $k > 0$ threads be represented by the matrices $T^{(i)} \in \mathcal{T}$, where each $T^{(i)}$ has order n. Then $||M_k||$ is bounded from above by $2k\,n^k$.*

Proof. We prove this lemma by induction on the definition of the Kronecker sum. For $k = 1$ the lemma is true. If we assume that for m threads $||M_m|| \leq 2m\,n^m$, then for $m+1$ threads $||M_{m+1}|| \leq 2m\,n^m \cdot n + n^m \cdot 2n = 2(m+1)\,n^{m+1}$. Thus, we have proved Lemma 5. □

Compared to the full matrix of order n^k with n^{2k} entries the resulting matrix has significantly fewer non-zero entries, namely $2k\,n^k$. By using the following definition we will prove that the matrices are sparse.

Definition 4 (Sparse Matrix). *We call a n-by-n matrix M sparse if and only if $||M|| = O\big(n\big)$.*

Lemma 6. *CFGs and RCFGs have Sparse Adjacency Matrices.*

Proof. Follows from Subsect. 2.2 and Def. 4. □

Lemma 7. *The Matrix P of a Program $\mathcal{P}$ is Sparse.*

Proof. Let $T = \bigoplus_{i=1}^{k} T^{(i)} \in \mathcal{M}$ be a N-by-N adjacency matrix of a program. We require that each of the k threads has order n in its adjacency matrix $T^{(i)}$. From Lemma 5 we know $||T|| = O\big(2k\,n^k\big)$. In addition, $N = n^k$ is given by Lemma 4. Hence, for k threads and by using Definition 4 we get $||T|| \leq 2k\,n^k = 2k\,N = O\big(N\big)$. A similar result holds for S and $P = T \circ S$. □

Lemma 7 enables the application of memory saving data structures and efficient algorithms. Algorithms may for example work on adjacency lists. Clearly, the space requirements for the adjacency lists are linear in the number of nodes. In the worst-case the number of nodes increases exponentially in the number of threads.

3.5 Efficient Implementation of the ∘-Operation

This subsection is devoted to an efficient implementation of the ∘-operation. First we define the Selective Kronecker product which we denote by $\oslash$. This operator synchronizes only identical labels $l \in \mathcal{L}_\mathrm{S}$ of the two input matrices.

Definition 5 (Selective Kronecker product). *Given two matrices A and B we call $A \oslash_L B$ their Selective Kronecker product. For all $l \in L \subseteq \mathcal{L}$ let $A \oslash_L B = (a_{i,j}) \oslash_L (b_{p,q}) = (c_{i.p,j.q})$, where*

$$c_{i.p,j.q} = \begin{cases} l & \text{if } a_{i,j} = b_{p,q} = l \wedge l \in L, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 6 (Filtered Matrix). *We call M_L a* Filtered Matrix *and define it as a matrix of order $o(M)$ containing entries $l \in L \subseteq \mathcal{L}$ of M and zeros elsewhere as follows.*

$$M_L = (m_{L;i,j}), \text{ where } m_{L;i,j} = \begin{cases} l & \text{if } m_{i,j} = l \wedge l \in L, \\ 0 & \text{otherwise.} \end{cases}$$

Note that

$$\sum_{l \in \mathcal{L}_\mathrm{S}} (T_l \otimes S_l) = T \oslash_{\mathcal{L}_\mathrm{S}} S. \tag{5}$$

In the following we use $o(S_{\mathcal{L}_\mathrm{V}}) = \prod_{i=1}^{r} o(S^{(i)}) = o(S)$. Note that S contains only labels $l \in \mathcal{L}_\mathrm{S}$. Hence, when the ∘-operator is applied for a label $l \in \mathcal{L}_\mathrm{V}$, we get $S_l = Z_{o(S)}$, i.e. a zero matrix of order $o(S)$. Thus we obtain $\sum_{l \in \mathcal{L}_\mathrm{V}} (T_l \oplus S_l) = T_{\mathcal{L}_\mathrm{V}} \otimes I_{o(S)}$. We will prove this below.

Finally, we can refine Eq. (4) by stating the following lemma.

Lemma 8. *The ∘-operation can be computed efficiently by*

$$P = T \circ S = T \oslash_{\mathcal{L}_S} S + T_{\mathcal{L}_V} \otimes I_{o(S)}.$$

Proof. Using Eq. (4) $P = T \circ S$ is given by $\sum\limits_{l \in \mathcal{L}_\mathrm{S}} (T_l \otimes S_l) + \sum\limits_{l \in \mathcal{L}_\mathrm{V}} (T_l \oplus S_l)$. According to Eq. (5) the first term is equal to $T \oslash_{\mathcal{L}_\mathrm{S}} S$. By mentioning $S_l = Z_{o(S)}$ for $l \in \mathcal{L}_\mathrm{V}$, Lemma 1, and Def. 2, the second term fulfills.

$$\sum_{l \in \mathcal{L}_\mathrm{V}} (T_l \oplus S_l) = \sum_{l \in \mathcal{L}_\mathrm{V}} \left(T_l \oplus Z_{o(S)}\right) = T_{\mathcal{L}_\mathrm{V}} \oplus Z_{o(S)} = T_{\mathcal{L}_\mathrm{V}} \otimes I_{o(S)}.$$

Note that S contains only $l \in \mathcal{L}_\mathrm{S}$. It is obvious that the non-zero entries of the first and the second term are $l \in \mathcal{L}_\mathrm{S}$ and $l \in \mathcal{L}_\mathrm{V}$, respectively. Both terms can be computed by iterating once through the corresponding sparse adjacency matrices, namely T and S. □

3.6 Lazy Implementation of Kronecker Algebra

Until now we have primarily focused on a pure mathematical model for shared memory concurrent systems. An alert reader will have noticed that the order of the matrices in our CPG increases exponentially in the number of threads. On the other hand, we have seen that the $\circ$-operation results in parts of the matrix $T \circ S$ that cannot be reached from the entry node of the underlying automaton (cf. Subsect. 3.3). This comes solely from the fact that synchronization excludes some interleavings.

Choosing a lazy implementation for the matrix operations, however, ensures that, when extracting the reachable parts of the underlying automaton, the overall effort is reduced to exactly these parts. By starting from the entry node and calculating all reachable successor nodes our lazy implementation exactly does this. Thus, for example, if the resulting automaton's size is linear in terms of the involved threads, only linear effort will be necessary to generate the resulting automaton.

Our implementation distinguishes between two kind of matrices: Sparse matrices are used for representing threads and semaphores. Lazy matrices are employed for representing all the other matrices, e.g. those resulting from the operations of the Kronecker algebra and our $\circ$-operation. Besides the employed operation, a lazy matrix simply keeps track of its operands. Whenever an entry of a lazy matrix is retrieved, depending on the operation recorded in the lazy matrix, entries of the operands are retrieved and the recorded operation is performed on these entries to calculate the result. In the course of this computation, even the successors of nodes are evaluated lazily. Retrieving entries of operands is done recursively if the operands are again lazy matrices, or is done by retrieving the entries from the sparse matrices, where the actual data resides.

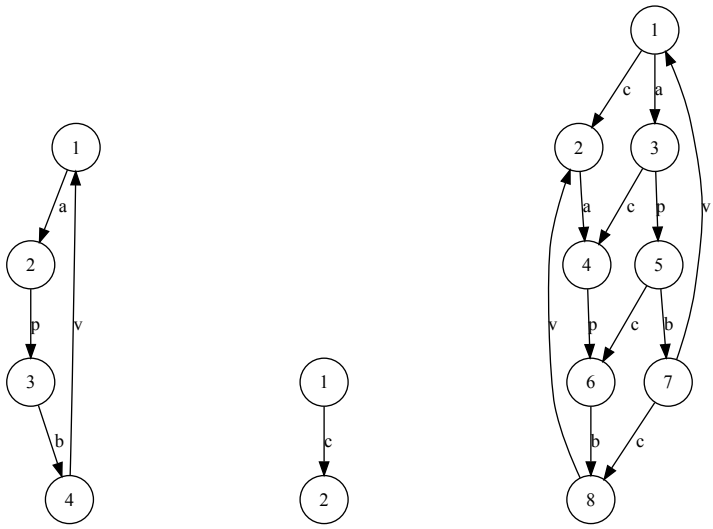
In addition, our lazy implementation allows for simple parallelizing. For example, retrieving the entries of left and right operands can be done concurrently. Exploiting this, we expect further performance improvements for our implementation if run on multi-core architectures.

3.7 Optimization for NSV

Our approach already works fine for practical settings. In this subsection we present additional optimizations which are optional.

As already mentioned in Subsect. 2.4 the Kronecker sum interleaves all entries. Sometimes this is disadvantageous because irrelevant interleavings will be generated if some basic blocks do not access shared variables. Such basic blocks can be placed freely as long as other constraints do not prohibit it.

For example consider the CFGs in Fig. 3. Assume for a moment that a, b, c, and d do not access shared variables. Then the overall behavior of the C-D-system can be described correctly by choosing *one* of the six interleavings depicted in Fig. 3(d), e.g., by $a \cdot b \cdot c \cdot d$. Hence the size of the CPG is reduced from nine nodes to five.



(a) E (b) F (c) $E \oplus F$

Fig. 7. A Counterexample

From now on we divide set $\mathcal{L}_{\mathrm{V}}$ into two disjoint sets $\mathcal{L}_{\mathrm{SV}}$ and $\mathcal{L}_{\mathrm{NSV}}$ depending on whether the corresponding basic blocks access shared variables or not.

The following example shows that NSV-edges cannot always be eliminated.

Example 4. In this example we use the graphs depicted in Fig. 7. The graphs E and F form the input graphs. It is assumed that a is the only edge not accessing a shared variable. All graphs have Node 1 as entry node. We show that it is not sufficient to chose exactly one NSV-edge. The matrix $E \oplus F$ is given by

$$\begin{pmatrix} 0 & c & a & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c & p & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & p & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c & b & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & b \\ v & 0 & 0 & 0 & 0 & 0 & 0 & c \\ 0 & v & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The graph represented by $E \oplus F$ which is structurally isomorph to $(E \oplus F) \circ S$ is depicted in Fig.7(c). Both loops in the CPG must be preserved. Otherwise the program would be modeled incorrectly. By removing an edge labeled by a, we would change the program behavior. Thus it is not sufficient to use only one edge labeled by a.

In general, the only way to reduce the size of the resulting CPG is by studying the matrix $T \circ S$. One way would be to output the automaton from $T \circ S$ and try to find reductions afterwards. We decided to perform such reductions during the output process such that a unnecessarily large automaton is not generated. It turned out that the problems to be solved to perform these reductions are hard. This will be discussed in detail below.

Fig. 8 shows the algorithm employed for the output process in pseudo code. By $\ell(n \to i)$ we denote the label assigned to edge $(n \to i)$. In short, the algorithm records all NSV-edges and proceeds until no other edges can be processed. Then it chooses one label of the NSV-edges. From the set of all recorded edges with this label a subset is determined such that all the edges in the subset can be reached from all nodes that have been processed so long. This is a necessary condition, if we want to eliminate the edges outside the subset. Determining a minimal subset under this constraint, however, is known as the *Set Covering Problem* which is NP-hard. We decided to implement a greedy algorithm. However, it turned out that in most cases we encountered a subset of size one, which trivially is optimal.

If no subset can be found, no edges can be eliminated.

Concerning Ex. 4 we note that the reason why none of the NSV-edges can be eliminated, can be found in the presence of the loop in E. Our output algorithm traverses the CPG in such a way that we do not know in advance if a loop will be constructed later on. Hence our algorithm has to be aware of loops that will be constructed in the future. This is done by remembering eliminated edges which will be reconsidered if a suitable loop is encountered.

In detail, if edges can be eliminated, we remember the set of eliminated edges $\mathcal{R}$ in set RECONSIDER together with a copy of the current set DONE. If later on we encounter a path in the CPG that reaches some nodes in this set DONE, we have to reconsider our decision. In this case all edges in $\mathcal{R}$ are reconsidered for being present in the CPG. Note that several RECONSIDER-sets can be affected if such a "backedge" is found. Note also that this reconsider mechanism handles Ex. 4 correctly.

Our implementation showed that the decision which label is chosen in Line 29 is also crucial. The number of edges (and nodes) being eliminated heavily depends on this choice. We are currently working on heuristics for this choice.

In the following we execute the algorithm on the example of Fig. 3 under the above conditions, i.e., a, b, c, and d do not access shared variables. At the beginning we have $\mathsf{TBD} = \{1\}$ and $\mathsf{TBDNSV}(a) = \mathsf{TBDNSV}(b) = \mathsf{TBDNSV}(c) = \mathsf{TBDNSV}(d) = \mathsf{DONE} = \emptyset$. Since RECONSIDER-sets are not necessary in this example, we do not consider them in the following to keep things simple.

The 1st iteration finds NSV-edges only. So: $\mathsf{TBDNSV}(a) = \{(1 \to 4)\}$, $\mathsf{TBDNSV}(c) = \{(1 \to 2)\}$, $\mathsf{DONE} = \{1\}$ and the other sets are empty.

The 2nd iteration chooses label a in Line 29. SUBSET clearly is $\{(1 \to 4)\}$, $\mathsf{TBD} = \{4\}$, and $\mathsf{TBDNSV}(a) = \emptyset$.

The 3rd iteration processes Node 4 and again finds NSV-edges only. So: $\mathsf{TBDNSV}(c) = \{(1 \to 2), (4 \to 5)\}$ and $\mathsf{TBDNSV}(b) = \{(4 \to 7)\}$. DONE becomes $\{1, 4\}$.

```
OutputCPG ()
1   TBD ← {startnode}
2   TBDNSV(ℓ ∈ L_NSV) {array of sets; all sets initialized to ∅}
3   DONE ← ∅
4   while TBD ≠ ∅ or ∃ℓ : TBDNSV(ℓ) ≠ ∅ do
5     if TBD ≠ ∅ then
6       n ← Element(TBD)   {choose one element of set TBD}
7       print n
8       for all edges (n → i) do
9         if ℓ(n → i) ∈ L_NSV then
10          TBDNSV(ℓ(n → i)) ← TBDNSV(ℓ(n → i))∪
11                                  {(n → i)}
12        else
13          TBD ← TBD ∪ {i}
14          print (n → i)
15        endif
16        while ∃R : i ∈ R and ∃D : {(D, R)} ∈ RECONSIDER do
17            {we have found a path back to a set of nodes
18            which we have used to eliminate NSV edges;
19            all these edges have now to be reconsidered}
20          for (m → j) ∈ R do
21            TBDNSV(ℓ(m → j)) ← TBDNSV(ℓ(m → j))∪
22                                  {(m → j)}
23          endfor
24          RECONSIDER ← RECONSIDER \ {(D, R)}
25        endwhile
26      endfor
27      DONE ← DONE ∪ {n}; TBD ← TBD \ DONE
28    else {TBD = ∅ }
29      ℓ ← NonEmptyElement(TBDNSV)
30        {choose one label with non-empty set in TBDNSV}
31      SUBSET ← SmallestSubset(TBDNSV(ℓ), DONE)
32        {choose smallest subset of TBDNSV(ℓ) such that
33        subset can be reached from all nodes in set DONE}
34      if TBDNSV(ℓ) \ SUBSET ≠ ∅ then
35        RECONSIDER ← RECONSIDER ∪
36                        {(DONE, TBDNSV(ℓ) \ SUBSET)}
37          {remember eliminated edges;
38          in case we find a path back to nodes in DONE,
39          we have to reconsider these edges}
40      endif
41      for (n → i) ∈ SUBSET do
42        print (n → i)
43        TBD ← TBD ∪ {i}
44      endfor
45      TBDNSV(ℓ) ← ∅
46    endif
47  endwhile
```

Fig. 8. Output CPG

The 4th iteration chooses label b in Line 29. Thus SUBSET clearly is $\{(4 \to 7)\}$, TBD $= \{7\}$, and TBDNSV$(b) = \emptyset$.

The 5th iteration processes Node 7 and finds one NSV-edge labeled c. So: TBDNSV$(c) = \{(1 \to 2), (4 \to 5), (7 \to 8)\}$. DONE becomes $\{1, 4, 7\}$.

The 6th iteration handles label c. The smallest subset is found to be $\{(7 \to 8)\}$ since Node 7 can be reached from each of the nodes in set DONE $= \{1, 4, 7\}$. Hence, edges $(1 \to 2)$ and $(4 \to 5)$ can be eliminated, i.e., they are not printed. So: TBDNSV$(c) = \emptyset$ and TBD $= \{8\}$.

The 7th iteration finds one NSV-edge labeled d. Thus we continue with TBDNSV$(d) = \{(8 \to 9)\}$. DONE becomes $\{1, 4, 7, 8\}$.

The 8th iteration handles label d. We obtain TBDNSV$(d) = \emptyset$ and TBD $= \{9\}$.

The 9th iteration prints Node 9, sets DONE $= \{1, 4, 7, 8, 9\}$ and TBD $= \emptyset$. The algorithm terminates and the result is depicted in Fig. 9.

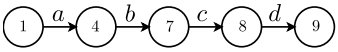


Fig. 9. Sequentialized C-D-System

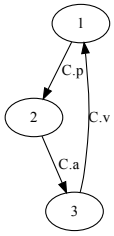
4 Client-Server Example

We have done analysis on client-server scenarios using our lazy implementation. For the example presented here we have used clients and a semaphore of the form shown in Fig. 10(a) and 10(b), respectively.

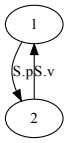
In Table 10(c) statistics for 1, 2, 4, 8, 16, and 32 clients are given. Fig. 10(d) shows the resulting graph for 8 clients. The few nodes in the resulting matrix and the node IDs indicate that most nodes in the resulting matrix are superfluous. The case of 32 clients and one semaphore forms a matrix with an order of approx. 3.706×10^{15}. Our implementation generated only 65 nodes in 0.43s. In fact we observed a linear growth in the number of clients for the number of nodes and edges and for the execution time. We did our analysis on an Intel Xeon 2.8 GHz with 8GB DDR2 RAM. Note that an implementation of the matrix calculus for shared memory concurrent systems has to provide node IDs of a sufficient size. The order of $T \circ S$ can be quite big, although the resulting automaton is small.

5 Generic Proof of Deadlock Freedom

Let S_i for $i \geq 1$ denote binary semaphores and let their operations be denoted by p_i and v_i.



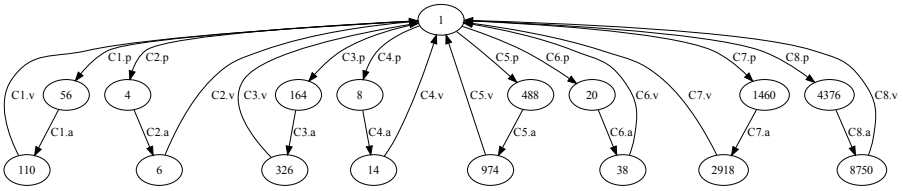
(a) Client



(b) Semaphore

Clients	Nodes	Edges	Exec. Time [s]	Potential Nodes
1	3	3	0.0013	6
2	5	6	0.0013	18
4	9	12	0.0045	162
8	17	24	0.0120	13,122
16	33	48	0.0680	86,093,422
32	65	96	0.4300	3.706×10^{15}

(c) Statistics



(d) Result for 8 Clients

Fig. 10. Client-Server Example

Definition 7. *Let $M = (m_{i,j}) \in \mathcal{M}$ denote a square matrix. In addition, let $\mathcal{P}_M = \{(i,j,r) \mid m_{i,j} = p_r \text{ for some } r \geq 1\}$ and $\mathcal{V}_M = \{(j,i,r) \mid m_{i,j} = v_r \text{ for some } r \geq 1\}$ (note the exchanged indexes (j,i)). We call M* p-v-symmetric *iff $\mathcal{P}_M = \mathcal{V}_M$.*

By definition of Kronecker sum and Kronecker product, it is easy to prove the following lemma.

Lemma 9. *Let M and N be p-v-symmetric matrices. Then $M \oplus N$, $M \otimes N$, and $M \circ N$ are also p-v-symmetric.* □

To be more specific, let $S_i = \begin{pmatrix} 0 & p_i \\ v_i & 0 \end{pmatrix}$ for $i \geq 1$. Then $S^{(r)} = \bigoplus_{i=1}^{r} S_i$ is p-v-symmetric.

Now, consider the p-v-symmetric matrix

$$M_k = \begin{pmatrix} 0 & p_1 & p_2 & \dots & p_k \\ v_1 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_k & 0 & 0 & \dots & 0 \end{pmatrix}.$$

Thus $M_k^{(n)} = \bigoplus_{i=1}^{n} M_k$ is also p-v-symmetric.

Now we state a theorem on deadlock freedom.

Theorem 1. *Let $P = M_k^{(n)} \circ S^{(k)}$ be the matrix of a n-threaded program with k binary semaphores, where $M_k^{(n)}$ and $S^{(k)}$ are defined above. Then the program is deadlock free.*

Proof. By definition and Lemma 9 P is p-v-symmetric. By Corollary 1 a deadlock manifests itself by a zero line, say ℓ, in matrix P. Since P is p-v-symmetric, column ℓ does only contain zeroes. Hence line ℓ is unreachable in the underlying automaton.

This clearly holds for all zero lines in P and thus the program is deadlock free. □

For counting semaphores we obtain matrices of the following type

$$\begin{pmatrix} 0 & p & 0 & \cdots & 0 & 0 \\ v & 0 & p & \cdots & 0 & 0 \\ 0 & v & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & p \\ 0 & 0 & 0 & \cdots & v & 0 \end{pmatrix}$$

which clearly is p-v-symmetric. Thus a similar theorem holds if counting semaphores are used instead of binary ones.

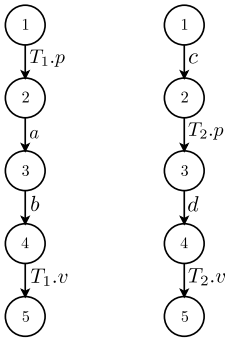
A short reflection shows that if we allow M_k to contain additional entries and non-zero lines and columns which do not contain ps and vs, the system is still deadlock free. So, we have derived a very powerful criterion to ensure deadlock freedom for a large class of programs, namely p-v-symmetric programs.

Concerning the example in Section 4 we note that if edges labeled a are removed from the clients, we obtain p-v-symmetric matrices. Thus this simple client-server system is deadlock free for an arbitrary number of clients. If we reinsert edges labeled a into the clients, no zero lines and columns appear (as noted above), so that the system is still deadlock free for an arbitrary number of clients.

Theorem 1 may be compared to the results of [8, 5], where for homogenous token passing rings it is proved that checking correctness properties can be reduced to rings of small sizes.

```
T1 ()
1   s.p              {edge T1.p}
2   r ← sv           {edge a}
3   r ← r + 1        {edge b}
4   sv ← r           {edge b}
5   s.v              {edge T1.v}
T2 ()
1   t ← sv           {edge c}
2   s.p              {edge T2.p}
3   t ← t + 1        {edge d}
4   sv ← t           {edge d}
5   s.v              {edge T2.v}
```

Fig. 11. Example Program



(a) T_1 (b) T_2

Fig. 12. RCFGs after Edge Splitting

6 A Data Race Example

We give an example, where a programmer is supposed to have used synchronization primitives in a wrong way. The program consisting of two threads, namely T_1 and T_2, and a semaphore s is given in Fig. 11. We assume that $sv = 0$ at program start. It is supposed that the program delivers $sv = 2$ when it terminates. Both threads in the program access the shared variable sv. The variables r and t are local to the corresponding threads. The programmer inadvertently has placed line 1 in front of line 2 in T_2.

After edge splitting we get the RCFGs depicted in Fig. 12. As usual the semaphore looks like Fig. 1(a). The corresponding matrices are

$$T_1 = \begin{pmatrix} 0 & T_1.p & 0 & 0 & 0 \\ 0 & 0 & a & 0 & 0 \\ 0 & 0 & 0 & b & 0 \\ 0 & 0 & 0 & 0 & T_1.v \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \text{ and } T_2 = \begin{pmatrix} 0 & c & 0 & 0 & 0 \\ 0 & 0 & T_2.p & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & T_2.v \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Although the following matrices are not computed by our lazy implementation, we give them here to allow the reader to see a complete example. To enable a concise presentation we define the following submatrices of order five:

$$H = \begin{pmatrix} 0 & c & 0 & 0 & 0 \\ 0 & 0 & T_2.p & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & T_2.v \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, I = \begin{pmatrix} T_1.p & 0 & 0 & 0 & 0 \\ 0 & T_1.p & 0 & 0 & 0 \\ 0 & 0 & T_1.p & 0 & 0 \\ 0 & 0 & 0 & T_1.p & 0 \\ 0 & 0 & 0 & 0 & T_1.p \end{pmatrix},$$

$$J = a \cdot I_5, K = b \cdot I_5, \text{ and } L = \begin{pmatrix} T_1.v & 0 & 0 & 0 & 0 \\ 0 & T_1.v & 0 & 0 & 0 \\ 0 & 0 & T_1.v & 0 & 0 \\ 0 & 0 & 0 & T_1.v & 0 \\ 0 & 0 & 0 & 0 & T_1.v \end{pmatrix}.$$

Now, we get $T = T_1 \oplus T_2$, a matrix of order 25, consisting of the submatrices defined above and zero matrices of order five (instead of Z_5 simply denoted by 0).

$$T = \begin{pmatrix} H & I & 0 & 0 & 0 \\ 0 & H & J & 0 & 0 \\ 0 & 0 & H & K & 0 \\ 0 & 0 & 0 & H & L \\ 0 & 0 & 0 & 0 & H \end{pmatrix}.$$

To shorten the presentation of $P = T \circ S$ we define the following submatrices of order ten:

$$U = \begin{pmatrix} 0 & 0 & c & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & T_2.p & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & d & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & T_2.v & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, V = \begin{pmatrix} 0 & T_1.p & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & T_1.p & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & T_1.p & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & T_1.p & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & T_1.p \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

$$W = a \cdot I_{10},\ X = b \cdot I_{10},\ \text{and } Y = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ T_1.v & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & T_1.v & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & T_1.v & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & T_1.v & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & T_1.v & 0 \end{pmatrix}.$$

With the help of zero matrices of order ten we can state the program's matrix

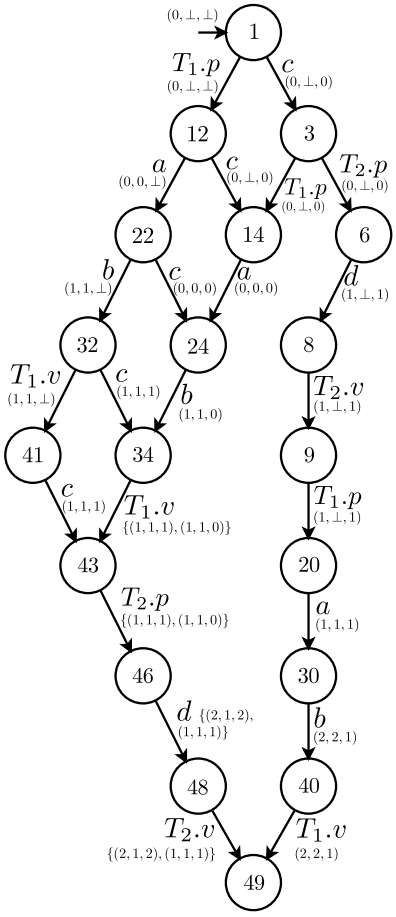


Fig. 13. Resulting CPG

$$P = T \circ S = T \circ \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix} = \begin{pmatrix} U & V & 0 & 0 & 0 \\ 0 & U & W & 0 & 0 \\ 0 & 0 & U & X & 0 \\ 0 & 0 & 0 & U & Y \\ 0 & 0 & 0 & 0 & U \end{pmatrix}.$$

Matrix P has order 50. The corresponding CPG is shown in Fig. 13. The lazy implementation computes only these 19 nodes. Due to synchronization the other parts are not reachable. In addition to the usual labels we have add a set of tuples to each edge in the CPG of Fig. 13. Tuple (x, y, z) denotes values of variables, such that $sv = x$, $r = y$ and $t = z$. We use $\perp$ to refer to an undefined value. A tuple shows the values after the basic block on the corresponding edge has been evaluated. The entry node of the CPG is Node 1. At program start we have the variable assignment $(0, \perp, \perp)$. At Node 49 we result in the set of tuples $\{(1,1,1),(2,1,2),(2,2,1)\}$. Due to the interleavings different tuples may occur at join nodes. This we reflect by a set of tuples. As stated above the program is supposed to deliver $sv = 2$. Thus the tuple $(1,1,1)$ shows that the program is erroneous. The error is caused by a data race between the edges c of thread T_2 and the edges a and b of thread T_1.

7 Empirical Data

In Sec. 4 we already gave some empirical data concerning client-server examples. In this section we give empirical data for ten additional examples.

Let $o(P)$ and $o(C)$ refer to the order of the adjacency matrix P, which is not computed by our lazy implementation, and the order of the adjacency matrix C of the resulting CPG, respectively. In addition k and r refer to the number of threads and the number of semaphores, respectively.

k	r	$o(P)$	$\sqrt{(o(P))}$	$o(C)$	Runtime [s]
2	4	256	16,00	12	0,03
3	5	4800	69,28	30	0,097542
4	6	124416	352,73	98	0,48655
3	6	75264	274,34	221	1,057529
4	7	614400	783,84	338	2,537082
4	8	1536000	1239,35	277	2,566587
4	8	737280	858,65	380	3,724364
4	13	298721280	17283,56	2583	96,024073
4	11	55050240	7419,58	3908	146,81
5	6	14929920	3863,93	7666	309,371395

Table 1. Empirical Data

In the following we use the data depicted in Table 1.[8] The numbers in the third column are rounded to two decimal places. As a first observation we note that except for one example all values of $o(C)$ are smaller as the corresponding values of $\sqrt{(o(P))}$. In addition, the runtime of our implementation shows a strong correlation to the order $o(C)$ of the adjacency matrix C of the generated CPG with a Pearson product-moment correlation coefficient of 0,9990277130. In contrast the values of the theoretical order $o(P)$ of the resulting adjacency matrix P correlates to the runtime only with a correlation coefficient of 0.2370050995.[9]

This observations show that the runtime complexity does not depend on the order $o(P)$ which grows exponentially in the number of threads. We conclude this section by stating that the collected data give strong indication that the runtime complexity of our approach is linear in the number of nodes present in the resulting CPG.

8 Related Work

Probably the closest work to ours was done by Buchholz and Kemper [3]. It differs from our work as stated in the following. We establish a framework for analyzing multithreaded shared memory concurrent systems which forms a basis for studying various properties of the program. Different techniques including dataflow analysis (e.g. [23–25, 14]) and model checking (e.g. [6, 9] to name only a few) can be applied to the generated CPGs. In this paper we use our approach in order to prove deadlock freedom. Buchholz and Kemper worked on generating reachability sets in composed automata. Our approach uses CFGs and semaphores to model shared memory concurrent programs. Buchholz and Kemper use it for describing networks of synchronized automata. Both approaches employ Kronecker algebra. An additional difference is that we propose optimizations concerning the handling of edges not accessing shared variables and lazy evaluation of the matrix entries.

In [9] Ganai and Gupta studied modeling concurrent systems for bounded model checking (BMC). Somehow similar to our approach the concurrent system is modeled lazily. In contrast our approach does not need temporal logic specifications like LTL for proving deadlock freedom for p-v-symmetric programs but on the other hand our approach may suffer from false positives. Like all BMC approaches [9] has the drawback that it can only show correctness within a bounded number of k steps.

Kahlon et al. propose a framework for static analysis of concurrent programs in [13]. Partial order reduction and synchronization constraints are used to reduce thread interleavings. In order to gain further reductions abstract interpretation is applied.

In [22] a model checking tool is presented that builds up a system gradually, at each stage compressing the subsystems to find an equivalent CSP process

[8] We did our analysis on an Intel Pentium D 3.0 GHz machine with 1GB DDR RAM running CentOS 5.6.

[9] Both correlation coefficients are rounded to ten decimal places.

with many less states. With this approach systems of exponential size ($\geq 10^{20}$) can be model checked successfully. This can be compared to our client-server example in Sect. 4, where matrices of exponential size can be handled in linear time.

Although not closely related we recognize the work done in the field of *stochastic automata networks* (SAN) which is based on the work of Plateau [19] and in the field of *generalized stochastic petri nets* (GSPN) (e.g. [4]) as related work. Compared to ours these fields are completely different. Nevertheless, basic operators are shared and some properties influenced this paper.

9 Conclusion

We established a framework for analyzing multithreaded shared memory concurrent systems which forms a basis for studying various properties of programs. Different techniques including dataflow analysis and model checking can be applied to CPGs. In addition, the structure of the matrices can be used to prove properties of the underlying program for an arbitrary number of threads. In this paper we used CPGs in order to prove deadlock freedom for the large class of p-v-symmetric programs.

Furthermore, we proved that in general CPGs can be represented by sparse matrices. Hence the number of entries in the matrices is linear in their number of lines. Thus efficient algorithms can be applied to CPGs.

We proposed two major optimizations. First, if the program contains a lot of synchronization, only a very small part of the CPG is reachable and, due to a lazy implementation of the matrix operations, only this part is computed. Second, if the program has only little synchronization, many edges not accessing shared variables will be present, which are reduced during the output process of the CPG. Both optimizations speed up processing significantly and show that this approach is very promising.

We gave examples for both, the lazy implementation and how we are able to prove deadlock freedom.

The first results of our approach (such as Theorem 1) and the performance of our prototype implementation are very promising. Further research is needed to generalize Theorem 1 in order to handle systems similar to the Dining Philosophers problem. In addition, details on how to perform (complete and sound) dataflow analysis on CPGs have to be studied.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, Massachusetts, 1986.
2. R. Bellman. *Introduction to Matrix Analysis.* Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 2nd edition, 1997.
3. P. Buchholz and P. Kemper. Efficient Computation and Representation of Large Reachability Sets for Composed Automata. *Discrete Event Dynamic Systems*, 12(3):265–286, 2002.

4. G. Ciardo and A. S. Miner. A Data Structure for the Efficient Kronecker Solution of GSPNs. In *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 22–31. IEEE Comp. Soc. Press, 1999.
5. E. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by Network Decomposition. In *Proc. 15th CONCUR, LNCS 3170*, pages 276–291. Springer, 2004.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
7. M. Davio. Kronecker Products and Shuffle Algebra. *IEEE Trans. Computers*, 30(2):116–125, 1981.
8. E. A. Emerson and K. S. Namjoshi. Reasoning about Rings. In *Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, POPL '95, pages 85–94, New York, NY, USA, 1995. ACM.
9. M. K. Ganai and A. Gupta. Efficient Modeling of Concurrent Systems in BMC. In *Proceedings of the 15th international workshop on Model Checking Software*, SPIN '08, pages 114–133, Berlin, Heidelberg, 2008. Springer-Verlag.
10. A. Graham. *Kronecker Products and Matrix Calculus with Applications*. Ellis Horwood Ltd., New York, 1981.
11. A. Hurwitz. Zur Invariantentheorie. *Math. Annalen*, 45:381–404, 1894.
12. W. Imrich, S. Klavzar, and D. F. Rall. *Topics in Graph Theory: Graphs and Their Cartesian Product*. A K Peters Ltd, 2008.
13. V. Kahlon, S. Sankaranarayanan, and A. Gupta. Semantic Reduction of Thread Interleavings in Concurrent Programs. In *TACAS'09: Proceedings of the 15th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505, pages 124–138, Berlin, Heidelberg, 2009. Springer-Verlag.
14. J. B. Kam and J. D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23:158–171, January 1976.
15. D. E. Knuth. *Combinatorial Algorithms*, volume 4A of *The Art of Computer Programming*. Addison-Wesley, 2011.
16. W. Kuich and A. Salomaa. *Semirings, Automata, Languages*. Springer-Verlag, 1986.
17. G. Küster. On the Hurwitz Product of Formal Power Series and Automata. *Theor. Comput. Sci.*, 83(2):261–273, 1991.
18. J. Miller. Earliest Known Uses of Some of the Words of Mathematics, Rev. Aug. 1, 2011. Website, 2011. Available online at `http://jeff560.tripod.com/k.html`; visited on Sept. 26th 2011.
19. B. Plateau. On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms. In *ACM SIGMETRICS*, volume 13, pages 147–154, 1985.
20. B. Plateau and K. Atif. Stochastic Automata Network For Modeling Parallel Systems. *IEEE Trans. Software Eng.*, 17(10):1093–1108, 1991.
21. G. Ramalingam. Context-Sensitive Synchronization-Sensitive Analysis Is Undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.
22. A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *Proc. of the First Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 133–152, London, UK, 1995. Springer-Verlag.
23. B. G. Ryder and M. C. Paull. Elimination Algorithms for Data Flow Analysis. *ACM Comput. Surv.*, 18(3):277–316, 1986.
24. B. G. Ryder and M. C. Paull. Incremental Data-Flow Analysis. *ACM Trans. Program. Lang. Syst.*, 10(1):1–50, 1988.

25. V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. A New Framework for Elimination-Based Data Flow Analysis Using DJ Graphs. *ACM Trans. Program. Lang. Syst.*, 20(2):388–435, 1998.
26. R. E. Tarjan. A Unified Approach to Path Problems. *Journal of the ACM*, 28(3):577–593, 1981.
27. J. G. Zehfuss. Ueber eine gewisse Determinante. *Zeitschrift für Mathematik und Physik*, 3:298–301, 1858.